View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Southern Denmark Research Output

Syddansk Universitet

Passive and Partially Active Fault Tolerance for Massively Parallel Stream Processing Engines

Su, Li; Zhou, Yongluan

Published in:
IEEE Transactions on Knowledge and Data Engineering (TKDE)

DOI:
10.1109/TKDE.2017.2720602

Publication date:
2017

Document version
Early version, also known as pre-print

Citation for pulished version (APA):
Su, L., & Zhou, Y. (2017). Passive and Partially Active Fault Tolerance for Massively Parallel Stream Processing Engines. IEEE Transactions on Knowledge and Data Engineering (TKDE), PP(99). DOI: 10.1109/TKDE.2017.2720602

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 09. Sep. 2018

Passive and Partially Active Fault Tolerance for Massively Parallel Stream Processing Engines

Li Su
University of Southern Denmark
lsu@imada.sdu.dk

Yongluan Zhou
University of Southern Denmark
zhou@imada.sdu.dk

arXiv:1508.04907v1 [cs.DC] 20 Aug 2015

ABSTRACT

Fault-tolerance techniques for stream processing engines can be categorized into passive and active approaches. A typical passive approach periodically checkpoints a processing task's runtime states and can recover a failed task by restoring its runtime state using its latest checkpoint. On the other hand, an active approach usually employs backup nodes to run replicated tasks. Upon failure, the active replica can take over the processing of the failed task with minimal latency. However, both approaches have their own inadequacies in Massively Parallel Stream Processing Engines (MPSPE). The passive approach incurs a long recovery latency especially when a number of correlated nodes fail simultaneously, while the active approach requires extra replication resources. In this paper, we propose a new fault-tolerance framework, which is Passive and Partially Active (PPA). In a PPA scheme, the passive approach is applied to all tasks while only a selected set of tasks will be actively replicated. The number of actively replicated tasks depends on the available resources. If tasks without active replicas fail, tentative outputs will be generated before the completion of the recovery process. We also propose effective and efficient algorithms to optimize a partially active replication plan to maximize the quality of tentative outputs. We implemented PPA on top of Storm, an open-source MPSPE and conducted extensive experiments using both real and synthetic datasets to verify the effectiveness of our approach.

1. INTRODUCTION

There is a recently emerging interest in building Massively Parallel Stream Processing Engines (MPSPE), such as Storm [2] and Infosphere[6], which make use of large-scale computing clusters to process continuous queries over fast data streams. Such continuous queries often run for a very long time and would unavoidably experience various system failures, especially in a large-scale cluster. As it is critical to provide continuous query results without significant downtime in many data stream applications, fault-tolerance techniques in Stream Processing Engines (SPEs) [4, 7, 25] have attracted a lot of attention.

Existing fault-tolerance techniques for SPEs can be generally categorized as passive and active approaches [13]. In a typical passive approach, the runtime states of tasks will be periodically extracted as checkpoints and stored at different locations. Upon failure, the state of a failed task can be restored from its latest checkpoint. While one can in general tune the checkpoint frequency to achieve trade-offs between the cost of checkpoint and the recovery latency, the checkpoint frequency should be limited to avoid high checkpoint overhead, which affects the system performance. Hence recovery latency is usually significant in a passive approach. When one wants to minimize the recovery latency as much as possible, it is often more efficient to use an active approach, which typically uses one backup node to replicate the tasks running on each processing node. When a node fails, its backup node can quickly take over with minimal latency.

Even though there are abundant fault-tolerance techniques in SPEs, developing an MPSPE [2, 6] poses great challenges to the problem. First of all, in a large cluster, there are often two different types of failures: independent failure and correlated failure [11, 20]. Previous studies mostly focused on independent failure that happens at a single node. Correlated failures are usually caused by failures of switches, routers and power facilities, and will involve a number of nodes failing simultaneously. With such failures, one has to recover a large number of failed tasks and temporally run them on an additional set of standby nodes before the failed ones are recovered. Using a passive fault tolerance approach, one has to keep the standby nodes running even their utilization is low most of the time in order to avoid the unacceptable overhead of starting them at recovery time. Furthermore, as checkpoints of different nodes are often created asynchronously, massive synchronizations have to be performed during recovery. Therefore it could be difficult to meet the user requirements on recovery latency even with a relatively high checkpoint frequency.

On the other hand, while an active fault-tolerance approach can achieve a lower recovery latency, it could be too costly for a large-scale computation. Consider a large-scale stream computation that is parallelized onto 100 nodes, one may not be able to afford another 100 backup nodes for active replication.

Another challenge is that there exist some time-critical applications which prefer query outputs being generated in good time even if the outputs are computed based on in-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

complete inputs. This kind of applications usually require continuous query output for real-time opportune decision-making or visualization. Consider a community-based navigation service, which collects and aggregates user-contributed traffic data in a real-time fashion and then continuously provides navigation suggestions to the users. Failure of some processing nodes could result in losing some user-contributed data. The system, while waiting for the failed nodes to recover, can continue to help drivers plan their routes based on the incomplete inputs. Other examples of such applications are like intrusion detections, online visualization of real-time data streams etc. Alerts of events matching the intrusion attack patterns or infographics generated over incomplete inputs are still meaningful to the users and should be generated without any major delay. Consider the long recovery latency for a large-scale correlated failure, the lack of trade-offs between recovery latency and result quality would not be able to fulfill the requirements of these applications.

To address the aforementioned challenges, we propose a new fault-tolerance scheme for MPSPEs, which is Passive and Partially Active (PPA). In a PPA scheme, a number of standby nodes will be used to prepare for recoveries from both independent and correlated failures. Checkpoints of the processing nodes will be stored at the standby nodes periodically. Rather than keeping them mostly idled as in a purely passive approach, we opportunistically employ them for active replications for a selected subset of the running tasks. In this way, we can provide very fast recovery for the tasks with active replicas. Furthermore, when the failed tasks contain those without active replicas, PPA provides *tentative* outputs with quality as high as possible. The results can then be rectified after the passive recovery process has been finished using similar techniques proposed in [4]. In general, PPA is more flexible in making use of the available resources than a purely active approach, and in the meantime can provide tentative outputs with a higher quality than a purely passive one.

In this paper, we focus on optimizing utilizing available resources for active replication in PPA, i.e. deciding which tasks should be included for active replication. In summary, we have made the following contributions in this paper:

(1) We present PPA, a passive but partially active fault-tolerance scheme for a MPSPE.

(2) As existing MPSPEs often involve user defined functions whose semantics are not easily available to the system, we propose a simple yet effective metric to estimate the quality of the tentative outputs.

(3) We propose an optimal dynamic programming algorithms and several heuristic algorithms to optimize the replication plan for a given query topology.

(4) We implement our approach in an open-source MPSPE, namely Storm [2] and perform an extensive experimental study on an Amazon EC2 cluster using both real and synthetic datasets. The results suggest that by adopting PPA, the accuracy of tentative outputs are significantly improved with limited amount of replication resources.

2. RELATED WORK

Fault-tolerance in SPE. Traditional fault-tolerance techniques for SPEs could be categorized as passive [13, 24, 18] and active approaches [13, 4, 12]. The technique of delta checkpoint [14] is used to reduce the size of checkpoints. The authors in [10] proposed techniques to reduce the checkpoint overhead by minimizing the sizes of queues between operators, which are part of the checkpoints. The work in [19] proposed to utilize the idle period of the processing nodes for active replication. Such optimizations are compatible to our PPA scheme and can be employed in our system.

For other large-scale computing systems, such like MapReduce [9] and Dryad [15], the overall job execution time is a critical metric. However, for MPSPEs, it is the end-to-end latency of tuple processing that matters, which makes the low-latency failure recovery an important feature in the context of MPSPEs. To reduce recovery latency, authors in [7, 25] proposed to use parallel recovery and/or integrating fault tolerance with scale-out operations. In parallel recovery, multiple tasks can be launched to recover a failed task and each of them is recovering a partition of the failed one to shorten the process of passive recovery. However, with a correlated failure, a large number of failed tasks need to be recovered simultaneously. Then the possibilities of fast scaling out and the degrees of parallel recovery would be constrained.

A hybrid fault-tolerance framework is proposed in [24] where operators can use different passive fault-tolerance techniques, including upstream buffering, local checkpoint and remote checkpoint. The authors propose an approach to minimize the total cost by choosing a fault-tolerance strategy for each operator. The work in [26] considers the "transient" failure caused by temporary workload spikes. Upon a transient failure is detected on a node, its active replica will be used to replace it and generate low-latency output until the transient failure is over. Different from these approaches, the trade-off of our work is between resource consumption and result accuracy with correlated failures.

Tentative Outputs. Borealis [4] uses active replication for fault tolerance and allows users to trade result latency for accuracy while the system is recovering from a failure. More specifically, if a failed node has no alive replica, Borealis will produce tentative outputs if the recovery cannot be finished within a user-defined interval. PPA adopts the similar mechanism for generating tentative outputs but explores more on optimizing the accuracy of tentative results.

Previous work [5] attempts to dynamically assign computation resources between primary computation and active replicas to achieve trade-offs between system throughput and fault-tolerance guarantee. In this work, tentative outputs will be produced if the failed tasks have no active replica. The optimization objective is to maximize the total number of tuples processed by the entire query topology with limited resources and an expensive brute-force algorithm is proposed to solve the problem. While PPA maximizes the effective input ratio which is roughly equal to the ratio of source data that can contribute to tentative outputs.

The work in [16] presented a fault injection-based approach to evaluate the importance of the computation units to the output accuracy, which only considered the independent failure. Zen [21] is another effort on optimizing operator placement within clusters under a correlated failure model, which specifies the probability that which nodes tend to fail together. The objective is to maximize the accuracy of tentative outputs after failures. As operator placement is orthogonal to the planning of active replications, their techniques can also be employed as a supplement to PPA.

Failure in Clusters. Previous studies have found that failure rates vary among different clusters and the number

of failures is in general proportional to the size of the cluster [22]. Correlated failures do exist and their scopes could be quite large [11, 20]. Therefore, considering correlated failure is inevitable for a MPSPE that is required to support low latency and nonstop computation.

3. SYSTEM MODEL

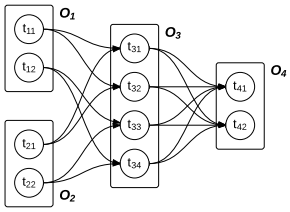


Figure 1: A topology that consists of 4 operators (O_1, O_2, O_3, O_4) with different numbers of tasks.

3.1 Data and Query Model

As in existing MPSPEs [2], we assume that a data item is modeled as a key-value pair. Without loss of generality, the key of a data item is assumed to be a string and the value is a blob in an arbitrary form that is opaque to the system.

A query execution plan in MPSPEs typically consists of multiple operators, each being parallelized onto multiple processing nodes based on the key of input data. Each operator is assumed to be a user-defined function. We model such query plan as a topology of the parallel tasks of all the query operators. By modeling each task as a vertex and the data flow between each pair of tasks as a directed edge, the query topology can be represented as a Directed Acyclic Graph (DAG). Figure 1 shows an example query topology. Each task represents the workload of an operator that is assigned to a processing node in the cluster and all the tasks that belong to the same operator will conduct the same computation.

An operator can subscribe to the outputs from multiple operators except for itself. The output stream of every task will be partitioned into a set of substreams using a particular partitioning function, which divides the keys of a stream into multiple key partitions and splits the stream into substreams based on these key partitions. For each task, the input substreams received from the tasks belonging to the same upstream neighboring operator will constitute an input stream. Therefore, the number of input streams of a task is up to the number of its upstream neighboring operators.

Similar to [27], we consider the following four common partitioning situations between two neighboring operators in a MPSPE. In the following descriptions, we consider an upstream operator containing N_1 tasks and a downstream operator containing N_2 tasks.

- *One-to-one*: each upstream task only sends data to a single downstream task and a downstream task only receives data from a single upstream task.
- *Split*: each upstream task sends data to M_2, $2 \leq M_2 < N_2$, downstream tasks and each downstream task only receives data from a single upstream task.
- *Merge*: each upstream task sends data to only one downstream task and each downstream task receives data from M_1, $2 \leq M_1 < N_1$, upstream tasks.
- *Full*: each upstream task sends data to all N_2 downstream tasks.

3.2 PPA Replication Plan

Given a topology T and its whole set of tasks $\mathcal{M}$, a PPA replication plan for T consists of two parts: a passive replication plan that covers all the tasks in $\mathcal{M}$ and a partially active replication plan which covers a subset of $\mathcal{M}$, denoted as $\mathbf{P}$. With the passive replication plan, checkpoints will be periodically created for all the tasks and stored at the standby nodes. For a task t_i, its checkpoint consists of t_i's computation state and output buffer. After a checkpoint is extracted from t_i, its upstream neighboring tasks will be notified to prune the unnecessary data from their output buffers. The buffer trimming should guarantee that, if t_i fails, its computation state can be recovered by loading its latest checkpoint and replaying the output buffers in its upstream tasks. On the other hand, for each $t_i \in \mathbf{P}$, an active replica will be created, which will receive the same input data and perform the same processing as t_i's primary copy.

Upon failures, the actively replicated tasks will be recovered immediately using their active replicas, meanwhile the tasks that are only passively replicated will be restored from their latest checkpoints. When there are some failed tasks belonging to $\mathcal{M} - \mathbf{P}$, tentative outputs will be produced before they are fully recovered. Such tentative outputs have a degraded quality due to the lost of input data that otherwise should be processed by the failed tasks belonging to $\mathcal{M} - \mathbf{P}$. We present how to optimize the partially active replication plan to maximize the quality of tentative outputs and the details of the system implementation in the following sections.

4. PROBLEM FORMULATION

4.1 Quality of Tentative Outputs

Previous works on load shedding [3, 17] have studied how to evaluate the quality of query outputs in case of the lost of input data. Their models assume full knowledge of the semantics of individual operators and hence can estimate the output quality in a relatively precise way. However, in existing MPSPEs, such as Storm, operators are often opaque to the system and may contain complex user-defined functions written in imperative programming languages. The existing models therefore cannot be easily applied. In our first attempt, we have tried to derive output accuracy models composed by some generic functions, which should be chosen or provided by the users according to the semantics of the operators. We found that this approach is not very user friendly and it may be very difficult for a user to provide such functions for a complicated operator.

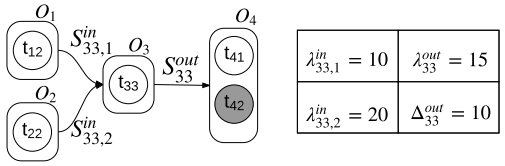
Therefore, we strive to design a model that requires users to provide minimum information of an operator's semantic, but yet is effective in estimating the quality of tentative outputs. More specifically, we propose a metric, called *Effective Input Ratio*(EIR), which is roughly equal to the ratio of source data that can contribute to the production of tentative outputs of a topology. This is based on the assumption that the accuracy of tentative outputs increases with more complete input and a PPA plan with a higher EIR value would incur a higher accuracy of tentative output. In the rest of this section, we will present a model to estimate the input loss of an operator, and then present the precise definition for EIR and the approach to estimate it.

4.1.1 Operator Input Loss Model

Suppose task t_{42} in Figure 1 fails, all its input streams cannot contribute to query outputs and hence we consider them effectively lost. To estimate the effective loss of input of the source operators (i.e. O_1 and O_2 in this example) caused by this failure, we have to propagate the input loss of t_{42} back to the source operators.

In the following descriptions, the set of input streams of task t_i, ($t_i \in O_t$), are denoted as $\{S^{in}_{i,1}, S^{in}_{i,2}, ..., S^{in}_{i,p}\}$, where the rate of $S^{in}_{i,j}$ is represented as $\lambda^{in}_{i,j}$. The rate of t_i's output stream, S^{out}_t, is referred to as λ^{out}_i, and its loss is denoted as Δ^{out}_i, where $\Delta^{out}_i \leq \lambda^{out}_i$. We propagate the loss of t_i's output stream to each of t_i's input streams, e.g. $S^{in}_{i,j}$, to calculate its input loss, e.g. $\Delta^{in}_{i,j}$.

Figure 2 depicts part of the topology in Figure 1 as well as the rates of the input and output streams of t_{33}. Δ^{out}_{33} represents the loss of output stream S^{out}_{33} incurred by the failure of task t_{42}. We distinguish two situations and use this example to illustrate the calculation of task input loss.



$\lambda^{in}_{33,1} = 10$	$\lambda^{out}_{33} = 15$
$\lambda^{in}_{33,2} = 20$	$\Delta^{out}_{33} = 10$

Figure 2: A part of the topology in Figure 1.

Correlated-Input Operator. O_t performs computations over the join results of its input streams. For example, suppose O_3 in Figure 2 is a correlated-input operator. Without further semantic information of O_3, we consider the effective input of t_{33} as the Cartesian product of its input streams, whose size is $\lambda^{in}_{33,1} \cdot \lambda^{in}_{33,2}$. By assuming that the losses of t_{33}'s input streams are proportional to their rates, and the loss of its output stream can be uniformly distributed among its effective inputs, we can deduce that: (1) the ratio between the losses of t_{33}'s input streams is equal to the ratio between their rates, i.e. $\Delta^{in}_{33,1} : \Delta^{in}_{33,2} = \lambda^{in}_{33,1} : \lambda^{in}_{33,1}$; (2) the relative loss of t_{33}'s effective input should be equal to the relative loss of its output, i.e. $1 - \frac{(\lambda^{in}_{33,1} - \Delta^{in}_{33,1}) \cdot (\lambda^{in}_{33,2} - \Delta^{in}_{33,2})}{\lambda^{in}_{33,1} \cdot \lambda^{in}_{33,2}} = \frac{\Delta^{out}_{33}}{\lambda^{out}_{33}}$. Finally, we have: $\Delta^{in}_{33,1} = 10 - \frac{10}{\sqrt{3}}$ and $\Delta^{in}_{33,2} = 20 - \frac{20}{\sqrt{3}}$. In summary, the losses of t_i's input streams can be calculated as follows:

$$\begin{cases} \lambda^{in}_{i,1} : \lambda^{in}_{i,2} : ... : \lambda^{in}_{i,p} = \Delta^{in}_{i,1} : \Delta^{in}_{i,2} : ... : \Delta^{in}_{i,p} \\ 1 - \prod_{j=1}^{p} \frac{(\lambda^{in}_{i,j} - \Delta^{in}_{i,j})}{\lambda^{in}_{i,j}} = \frac{\Delta^{out}_i}{\lambda^{out}_i} \end{cases}$$

If O_t is a correlated-input operator and $t_i \in O_t$ lost all of its output, we consider t_i as losing all of its effective input, that is, $\Delta^{in}_{i,j} = \lambda^{in}_{i,j}$, for any input stream $S^{in}_{i,j}$ of t_i.

Independent-Input Operator. O_t does not compute joins over input streams. If O_3 in Figure 2 is an independent-input operator, we assume that the data losses of t_{33}'s input streams are proportional to their rates, and its total input loss ratio is equal to its total output loss ratio. Then we have, in this example, $\Delta^{in}_{33,1} : \Delta^{in}_{33,2} = \lambda^{in}_{33,1} : \lambda^{in}_{33,2}$ and $\frac{\Delta^{in}_{33,1} + \Delta^{in}_{33,2}}{\lambda^{in}_{33,1} + \lambda^{in}_{33,2}} = \frac{\Delta^{out}_{33}}{\lambda^{out}_{33}}$. We can get: $\Delta^{in}_{33,1} = \frac{20}{3}$ and $\Delta^{in}_{33,2} = \frac{40}{3}$. Therefore, the losses of t_i's input streams can be calculated as follows:

$$\begin{cases} \lambda^{in}_{i,1} : \lambda^{in}_{i,2} : ... : \lambda^{in}_{i,p} = \Delta^{in}_{i,1} : \Delta^{in}_{i,2} : ... : \Delta^{in}_{i,p} \\ \frac{\sum_{j=1}^{p} \Delta^{in}_{i,j}}{\sum_{k=1}^{p} \lambda^{in}_{i,k}} = \frac{\Delta^{out}_i}{\lambda^{out}_i} \end{cases}$$

Recall that one of the design principles is to request as little information of the operators' semantics as possible. We distinguish the aforementioned two types of operators simply because their characteristics for the calculation of effective input ratio are very different. With such information, the metric can be estimated much more precisely.

The output stream of an operator may be shared by multiple downstream operators. Suppose D_t denotes the set of downstream neighboring operators of O_t, then the task failure in any operator belonging to D_t will result in output loss to tasks in O_t. Denoting Δ^d_i as the loss of t_i's output stream caused by task failure in downstream operator O_d, the final output loss of t_i is calculated as:

$$\Delta^{out}_i = \frac{\sum_{O_d \in D_t}^{} \Delta^d_i}{|D_t|} \quad (1)$$

By denoting the set of tasks of an operator O_t as $\{t_1, t_2, ..., t_{M_t}\}$ and the losses of t_i's input streams as $\Delta^{in}_{i,1}, \Delta^{in}_{i,2}, ...\Delta^{in}_{i,p}$, the input loss ratio of O_t, ILR_t, is defined as:

$$ILR_t = \frac{\sum_{i=1}^{M_t} \sum_{j=1}^{p} \frac{\Delta^{in}_{i,j}}{\lambda^{in}_{i,j}}}{M_t \cdot p} \quad (2)$$

Note that input stream $S^{in}_{i,j}$ may consist of multiple substreams that are sourced from the tasks of its source operator. To calculate the output losses for the source tasks of the substreams in $S^{in}_{i,j}$, $\Delta^{in}_{i,j}$ will be split and assigned to the substreams in $S^{in}_{i,j}$ proportionally according to their rates.

4.1.2 Effective Input Ratio

In this subsection, we present how to estimate the effective input ratios of the source operators.

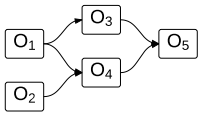


Figure 3: Correlated input operator: O_4; Independent input operator: O_1, O_2, O_3, O_5.

An example topology is depicted in Figure 3. One can see that the outputs of O_1 is shared by O_3 and O_4. Furthermore, as O_4 is assumed to be a correlated-input operator, the output streams of O_1 and O_2 are correlated and the tasks in O_4 cannot produce outputs if one of its input streams is lost. To take the sharing and correlation of output streams into consideration, the effective input ratio (EIR) of a topology T is defined in the form of a recursive function which starts from each sink operator, traverses through the whole topology and stops at the source operators.

In the following, M_t denotes the number of tasks in O_t and U_t refers to the set of upstream operators of O_t in T. Note that U_t is an empty set if O_t is a source operator. The EIR function is defined recursively as follows:

$$EIR_T = f_{eir}(O_s), \quad O_s \textit{ is the sink operator of } T.$$

If O_t is an independent-input operator:

$$\begin{cases} f_{eir}(O_t) = \frac{\sum^{O_u \in U_t} f_{eir}(O_u)}{|U_t|}, \ U_t \neq \emptyset \\ f_{eir}(O_t) = 1 - ILR_t, \ U_t \equiv \emptyset \end{cases}$$

If O_t is a correlated-input operator:

$$\begin{cases} f_{eir}(O_t) = \prod^{O_u \in U_t} f_{eir}(O_u),\ U_t \neq \emptyset \\ f_{eir}(O_t) = 1 - ILR_t,\ U_t \equiv \emptyset \end{cases}$$

Taking the topology in Figure 3 as an example, the EIR recursive function starts from the sink operator, O_5:

$$f_{eir}(O_5) = \frac{f_{eir}(O_3) + f_{eir}(O_4)}{2} = \frac{f_{eir}(O_1) + f_{eir}(O_1) \cdot f_{eir}(O_2)}{2}$$

As one can see, the recursion stops at source operators O_1 and O_2. The values of $f_{eir}(O_1)$ and $f_{eir}(O_2)$ can be calculated with their input loss ratios (defined in Eqn. 2) respectively. The shared source operator, O_1, appears two times in the final equation. As O_4 is a correlated-input operator, the output streams from O_1 and O_2 are correlated, which is expressed as multiplication of the EIR of O_1 and O_2 in the above equation.

4.2 Problem Statement

Before presenting the problem definition, we introduce a concept: *Minimal Complete Tree*, which is also referred to as *MC-tree* for simplicity in the following sections.

Definition 1. MINIMAL COMPLETE TREE (MC-TREE)*: A minimal complete tree is a tree-structured subgraph of the topology DAG. The source vertices of this subgraph correspond to tasks from the source operators and its sink vertex is a task from an output operator. A minimal complete tree can continuously contribute to propagating source input data if and only if all its tasks are alive.*

Taking the topology in Figure 1 for instance, if O_3 is an independent-input operator, tasks in $\{t_{11}, t_{31}, t_{41}\}$ can constitute an MC-tree and there are in total 16 MC-trees in the topology. However, if O_3 is a correlated-input operator, t_{31} cannot produce any output if either t_{11} or t_{21} fails. Hence tasks in $\{t_{11}, t_{21}, t_{31}, t_{41}\}$ can constitute an MC-tree and the number of MC-trees in the topology is equal to 8.

Based on Definition 1, if failures of tasks in an MC-Tree occur, it will only continue propagating source input data to final query output if and only if all of its failed tasks are actively replicated. Suppose Topology T consists of a set of operators $O_1, O_2, ..., O_N$ and the available resources can be used to actively replicate R ($R \leq |\mathcal{M}|$, $\mathcal{M}$ is the task set of T) tasks, then the problem of optimizing a partially active replication plan is defined as follows:

Definition 2. PARTIALLY ACTIVE PLAN*: Given a query topology T, choose R tasks for active replication such that, the EIR of the partial topology that is composed of the actively replicated MC-trees in T is maximized.*

We prove that the above problem is NP-hard. The detailed proof can be found in [23].

5. ACTIVE REPLICATION OPTIMIZATION

Recall that we consider the worst case scenario for a correlated failure, i.e. there is at least one failed task in every MC-tree. Before the completion of the passive recovery process, only the MC-trees whose failed tasks are actively replicated can produce tentative outputs. The optimization objective is to maximize the value of EIR with limited amount of resources used for active replication.

Algorithm 1: Dynamic Programming: PLANCORRELATEDFAILURE(R)

```
   Input: Amount of available resources R;
   Output: Replication plan P;
1  CP_0 ← ∅; usage ← 0; SC ← {CP_0};
   /* CP_0:initial replication plan; SC:candidate plan
      set;                                              */
2  while usage++ < R do
3      foreach candidate plan CP_i in SC do
4          dif ← usage − |CP_i|;
           /* |CP_i| is the number of replicated tasks in
              CP_i and dif is the number of tasks that
              can be added to CP_i at this step;        */
5          UT_i ← { MC-tree tr | tr ∉ CP_i};
6          u_i ← max{nonrep_tasks(tr, CP_i) | tr ∈ UT_i};
           /* nonrep_tasks(tr, CP_i) returns the number of
              non-replicated tasks of MC-tree tr in CP_i;
              */
7          if dif ≤ u_i then
8              foreach MC-tree tr_j ∈ {tr | tr ∉
               CP_i & nonrep_tasks(tr_j, CP_i) == dif} do
9                  CP_j ← CP_i ∪ tr_j;
10                 if CP_j ∉ SC then
11                     add CP_j to SC;
12         else Remove CP_i from SC;
13 P ← the candidate plan in SC with the maximal EIR value.
   Return P;
```

5.1 Dynamic Programming

We first present a dynamic programming algorithm that can generate an optimal replication plan for correlated failure. As has been introduced in section 4.2, we take MC-tree as the basic unit for replication candidates in the algorithm. Details of this algorithm are presented in Algorithm 1. It is essentially a bottom-up dynamic programming algorithm. We incrementally increase the number of resources to be used for active replication and enumerate the possible expansions of the plans produced in the previous step. Assuming the minimum size of MC-trees is r, one can obtain the first set of replication plans, referred to as SC, by replicating r tasks. At this step, each plan in SC contains exactly one MC-tree. Note that the MC-trees that have not been added to a candidate plan CP_i may also have replicated tasks if they share some tasks with another MC-tree within CP_i.

At the next iteration of the while loop starting at line 2, we increase the resource usage by 1. We scan through each candidate plan $CP_i \in SC$ to see if there is an MC-tree $tr_j \notin CP_i$ that contains a number of non-replicated tasks which is equal to $usage - |CP_i|$, where $|CP_i|$ is the number of replicated tasks in CP_i. For each MC-tree satisfying this condition, we create a new candidate plan CP_j (line 9) such that $CP_j \leftarrow CP_i \cup tr_j$. If CP_j has no duplicate in SC, then it will be inserted into SC. The algorithm will continue until *usage* is equal to the limit R.

The cost of scanning through SC can be reduced by removing a candidate plan CP_i from SC if all its possible expansions have been considered. More precisely, remove CP_i from SC if the maximum number of non-replicated tasks of the MC-trees not included in CP_i is less than the difference between the available resource at the current iteration, i.e. *usage*, and the current number of replicated tasks in CP_i (lines 7 and 12). After the while loop is finished, the candidate plan with the maximal EIR in SC will be returned.

The upper bound of the complexity of this algorithm is

Algorithm 2: GREEDY(R)

Input: The amount of available resources R;
Output: Replication plan $\mathbf{P}$

Initialize: $AS \leftarrow \emptyset$;
foreach *Task* $t_i \notin \mathbf{P}$ **do**
  $A_i \leftarrow$ the value of EIR if only t_i fails;
  $AS \leftarrow AS \cup \{A_i\}$;
Sort AS in ascending order;
$TS \leftarrow$ set of tasks whose corresponding EIR values are among top-R in AS;
$\mathbf{P} \leftarrow \mathbf{P} \cup TS$;
return $\mathbf{P}$

$O\left(2^{\mathcal{T}}\right)$, where $\mathcal{T}$ is the number of MC-trees in the query topology, which varies with the topology structures and has an upper bound of $O(M^N)$, where N is the number of operators and M is the average degree of parallelization of operators in T.

The following theorem states the optimality of this dynamic programing algorithm, which is proven in [23].

Theorem 1. *Let* $\mathbf{P}$ *be the replication plan with EIR* $\mathbf{E}$ *produced by Algorithm 1 and* $\mathbf{P_t}$ *be a different replication plan with EIR* $\mathbf{E}_t \geq \mathbf{E}$. *The resource usage of* $\mathbf{P}$ *is always equal to or smaller than that of* $\mathbf{P_t}$.

5.2 Greedy Algorithm

We present a greedy algorithm. For each task in the topology, the greedy algorithm will calculate the EIR of the topology by only failing this task. A task whose failure would lead to a smaller EIR will be assigned a higher priority for replication. We present the details of this greedy algorithm in Algorithm 2, which will first rank all the tasks in ascending order based on the EIR calculated by their respective failures. Then it will iterate to choose the corresponding task that would cause the minimal EIR among all the remaining non-replicated tasks in the set AS.

The complexity complexity of the greedy algorithm is equal to $O(N \cdot M)$. where the notations have been explained in Section 5.1. Although this complexity is much lower than that of the dynamic programming algorithm, it fails to consider whether the tasks in the replication plan could form complete MC-trees, which will damage its performance especially when the number of active replicated tasks is small. The experimental results in section 7.2 can verify this defect of the greedy algorithm.

5.3 Structure-Aware Algorithm

The dynamic programming algorithm searches for the optimal plan by selecting a subset of MC-trees for replication under the resource constraint to maximize the value of EIR. Inspired by this, we design a structure-aware algorithm that, at each step, rather than enumerating all the possible expansions of a candidate plan, only expands it with an MC-tree that can incur the greatest increase in EIR per resource unit.

Unfortunately, even such a greedy approach may fall short under the following situation. Consider a topology T that consists of a sequence of k operators and all the operators use Full partitioning, the number of MC-trees within T is equal to $\prod_{i=1}^{k} M_i$, where M_i is the number of tasks of operator O_i. In such a topology, the number of MC-trees will grow very fast with increasing number of operators. Therefore, even a greedy search among the possible combinations of MC-trees would not perform well.

Algorithm 3: PLANSTRUCTUREDTOPOLOGY($\mathbf{P}, R, T$)

Input: An initial plan $\mathbf{P}$; The amount of available resources R; Topology T;
Output: Replication plan $\mathbf{P}$;

$usage = 0$; $S_u \leftarrow$ Set of the units split from topology T;
foreach *Unit* $U_i \in S_u$ **do**
  Build segment set G_i;
while $usage \leq R$ **do**
  $Candidates \leftarrow \emptyset$;
  foreach *Unit* $U_i \in S_u$ **do**
    foreach *non-replicated segment* $g_i \in U_i$ **do**
      $CG_i \leftarrow \{g_i\}$;
      if $EIR_{\mathbf{P}} = EIR_{\mathbf{P} \cup CG_i}$ **then**
        Conduct a BFS from U_i to traverse all the units:
        foreach *visited unit* U_j *during the BFS* **do**
          Segment $g_j \leftarrow$ **max_eir** (U_j) ;
          `/* max_eir(U_j) returns the segment in U_j, which is connected with segment in CG_i and has the maximal EIR with U_j treated as an independent topology; */`
          if $|CG_i| + |g_j| \leq usage$ **then**
            $CG_i = CG_i \cup g_j$;
          else Stop the BFS;
      $Candidates \leftarrow Candidates \cup CG_i$;
  Find CG_{opt} from $Candidates$ such that the following value is maximized: $(EIR_{\mathbf{P} \cup CG_{opt}} - EIR_{\mathbf{P}})/|CG_{opt}|$;
  $\mathbf{P} = \mathbf{P} \cap CG_{opt}$; $usage = usage + |CG_{opt}|$;
  if $CG_{opt} \neq \emptyset$ **then** return $\mathbf{P}$;
  Remove the completely replicated units from S_u;
Return $\mathbf{P}$;

To solve this problem, we will firstly decompose a general topology into two specific types of topologies, namely full topologies and structured topologies, and then optimize them separately. The definitions of these two types of topologies are as follows:

- **Structured topology** is defined as a topology where only the operators, that produce outputs of this topology, can have a Full partitioning function and the others have other types of partitioning functions.
- **Full topology** is defined as a topology that all of its operators have a Full partitioning function.

The rest of this section is organized as follows: firstly, we present a structure-aware algorithm for structured topologies and full topologies respectively. Then we will explain how to generate a partially active replication plan by decomposing a general topology into several sub-topologies, each being either a structured topology or a full topology.

5.3.1 Algorithm for Structured Topology

Although we define structured topology such that Full partitioning only exists in the output operators, the number of MC-trees in a structured topology could still be very large. Consider the situation that a task t_i receives N_{in} input streams and produce N_{out} output streams, there will be $N_{in} * N_{out}$ MC-trees containing t_i. In addition, if t_i joins N_k substreams from operator O_k with N_j substreams from operator O_j, the number of MC-trees containing t_i will at least be equal to $N_k \cdot N_j$. To avoid bad performance due to the large number of MC-trees, we split a structured topology into multiple units. The split is done such that, within a

unit, the number of MC-trees is equal to the maximal number of input substreams among the operators of this unit. We refer to an MC-tree in a unit as *segment* to differentiate it from the concept of a complete MC-tree in the topology.

The situation of multiple input streams and multiple output streams occurs on the task who has an input stream partitioned with Merge and an output stream partitioned with Split, a unit boundary will be set between this operator and its upstream neighboring operator that uses Merge partitioning. For instance, a unit boundary will be set between O_1 and O_3 in the topology in Figure 4. The situation that a task joins multiple input substreams from one operator with substreams from other operators happens on the tasks of join operators that have at least one input stream partitioned with Merge partitioning. A unit boundary will be set between the join operator and its upstream neighboring operator whose output stream is partitioned with a Merge partitioning function. If O_3 in Figure 4 is a join operator, there will be a boundary between O_3 and each of its upstream neighboring operators.

Note that, with such a decomposed topology, replicating a segment is beneficial only if all the other segments within the same complete MC-tree are also replicated. In other words, we should avoid enumerating plans that replicate a set of disconnected segments.

The details of the structure-aware algorithm for structured topology are presented in Algorithm 3. The algorithm searches through the units generated from input topology. Within unit U_i, if the set of non-replicated segments is not empty, we check whether replicating these segments will increase the final output accuracy (line 9). Note that this will only be true if this segment can form a complete MC-tree with the other replicated segments within the current plan. Each of such segments will be put into a candidate pool (line 16). If the segment g_i does not enhance the plan's EIR, we conduct a BFS (Breadth-first search) starting from U_i and traversing through all the units in Topology T. The BFS is terminated until *usage* is less then the non-replicated tasks in CG_i. Finally, every unit visited during the BFS contributes a segment to CG_i and the segments from neighboring units are connected (lines $10-15$). Then we put such a set of segments as one candidate in the candidate pool.

After finishing the scanning of all units, we get a candidate pool consisting of a number of segment sets, each containing one or more segments. We use a profit density function to rank the candidates. The profit density of a candidate CG_k is calculated as $(EIR_{P\cup CG_k} - EIR_P)/|CG_k|$, where EIR_P is the EIR value of plan P, $EIR_{P\cup CG_k}$ is the EIR value after expanding P by replicating segment in CG_k. $|CG_K|$ is the number of non-replicated tasks within CG_k. The plan in the candidate pool with the maximum profit density will be merged with the input plan P and returned. The complexity of Algorithm 3 is equal to $O(R \cdot N \cdot M^2 \cdot E)$, where R is the amount of available replication resources, N is the number of operators, M represents the average degree of parallelization of operators in T, and E is the number of neighboring unit pairs.

5.3.2 Algorithm for Full Topology

Each task within a full topology will send input data to all the tasks that belong to its downstream neighboring operators. We propose an algorithm for full topology as illustrated in Algorithm 4. The basic idea of this algorithm

Algorithm 4: PLANFULLTOPOLOGY($\mathbf{P}$, R, T)

Input: Initial replication plan $\mathbf{P}$; Amount of available resources R; Topology T;
Output: Replication Plan $\mathbf{P}$

1 Initialize: $usage \leftarrow 0$;
2 $N \leftarrow$ *Number of operators*;
3 Sort the set of tasks S_i of each operator O_i based on the EIR increase, δ_{ij} , of tasks;
4 **if** $P = \emptyset$ & $N \leq R$ **then**
5 **foreach** O_i **do**
6 Let p_{ik} be the node in S_i that has the largest EIR increase δ_{ik};
7 $\mathbf{P} \leftarrow \mathbf{P} \cup \{p_{ik}\}$; $S_i \leftarrow S_i - \{p_{ik}\}$;
8 $usage = N$;
9 **if** $\mathbf{P} = \emptyset$ & $N > R$ **then** return $\mathbf{P}$;
10 **while** $usage < R$ **do**
11 $Candidates \leftarrow \emptyset$;
12 **foreach** O_i **do**
13 Let p_{ik} be the node in S_i that has the largest EIR increase δ_{ik};
14 $Candidates \leftarrow Candidates \cup P_i \cup \{p_{ik}\}$;
15 $P_j \leftarrow$ **max_accuracy_plan**($Candidates$);
16 $S_j \leftarrow S_j - \{p_{jk}\}$; $\mathbf{P} \leftarrow P_j$; $usage++$;
17 Return $\mathbf{P}$;

is that, within any operator, we always prefer to replicate the task that will bring the maximum increase of EIR under the assumption that all the other tasks that belong to the same operator are failed and the tasks that belong to other operators are alive. We denote the increase of EIR by replicating task t_{ij} as δ_{ij}. If the input plan P is empty, we first select one task from each operator that has the largest δ_{ij} among all the tasks in this operator and put it into P (lines $4-7$). If P is not empty, we iterate and select R tasks that have larger EIR increase, i.e. δ_{ik} , than other tasks in the topology and put them into P (lines $11-17$). The complexity of this algorithm is $O(N \cdot R)$, where R is the amount of available replication resources and N is the number of operators.

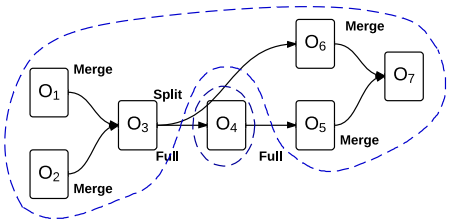


Figure 4: Example of splitting a topology

5.3.3 Solution for General Topology

With the above algorithms for specific topology structures, we divide a general topology into several sub-topologies and then use the corresponding algorithms according to the type of each sub-topology to generate the replication plans. We require that at least one partitioning function between any two neighboring sub-topologies is Full and the amount of sub-topologies is minimized. The reason behind this requirement is to make the selection of the replication segments in the sub-topologies independent from each other.

The split algorithm explores the topology using multiple depth-first searches (DFS). At the beginning, only the output operator of the given topology is in the start point set SP. At each iteration, we will pick an operator, O_s, from

Algorithm 5: PLANGENERALTOPOLOGY(R,T)

Input: The amount of available resources R; Topology T;
Output: Partial replication plan $\mathbf{P}$;
1 Initialize: decompose the complete topology T into sub-topologies: $TS_1, TS_2, \ldots$;
2 $\mathbf{P} \leftarrow \emptyset$, $S_A \leftarrow \emptyset$, $usage \leftarrow 0$;
3 **if** $R <$ *Number of operators in* T **then**
4 | Return $\mathbf{P}$;
5 **foreach** *Sub-Topology* TS_i **do**
6 | $N_i \leftarrow$ Number of operators in TS_i;
7 | $P_i \leftarrow$ `PlanSubTopology` $(\emptyset, R_i, TS_i)$; $\mathbf{P} \leftarrow \mathbf{P} + P_i$;
8 | $P_i' \leftarrow$ `PlanSubTopology` (P_i, R_i, TS_i);
9 | $C_i \leftarrow |P_i'| - |P_i|$; $\Delta_i \leftarrow \frac{EIR_{P_i'} - EIR_{P_i}}{C_i}$;
10 | Put Δ_i into S_A in descending order; $usage += N_i$;
11 **while** $usage < R$ **do**
12 | $LastUsage \leftarrow usage$; $j \leftarrow 1$;
13 | **while** $j \leq |S_A|$ **do**
14 | | $\Delta_i \leftarrow j$th value in S_A; $j++$;
15 | | **if** $C_i + usage \leq R$ **then**
16 | | | Use P_i' to replace P_i in $\mathbf{P}$;
17 | | | Calculate new C_i, Δ_i. Insert Δ_i into S_A in descending order; break;
18 | **if** $usage = lastUsage$ **then** break;
19 Return $\mathbf{P}$;
Function: `PlanSubTopology`($\mathbf{P}$, N_i, T)
20 **if** T *is a full topology* **then**
21 | $\mathbf{P} \leftarrow$ PLANFULLTOPOLOGY($\mathbf{P}, N_i, T$);
22 **else** $\mathbf{P} \leftarrow$ PLANSTRUCTUREDTOPOLOGY($\mathbf{P}, N_i, T$);

SP and build a sub-topology by performing a DFS starting from O_s. If the DFS arrives at an operator O_i whose partitioning function is incompatible with the type of the current sub-topology, it will not further traverse O_i's downstream operators and O_i will not be added to the current sub-topology but instead be put into SP. Finally the algorithm will terminate until SP is empty. Figure 4 presents an example general topology, which is decomposed into 2 sub-topologies: $\{O_1, O_2, O_3, O_5, O_6, O_7\}$ and $\{O_4\}$.

We present details of the correlated-failure optimization algorithm for a general topology in Algorithm 5. The algorithm first decomposes the topology into sub-topologies which are either full topologies or structured topologies. Then the algorithm runs in multiple iterations. Within each iteration, it will try to get a replication plan from each sub-topology and select the one with the maximum profit density (lines $11 - 17$). The loop will be terminated when there is no more resource to replicate a complete MC-tree. The algorithm's complexity is equal to $O(R \cdot N \cdot M^2 \cdot E)$, where the notations have been explained in Section 5.3.1.

6. SYSTEM IMPLEMENTATION

In this section, we present the framework and implementation details of our system.

6.1 Framework

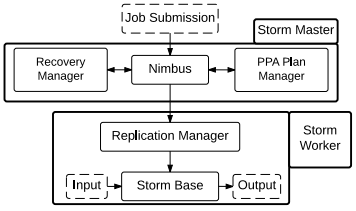


Figure 5: System Framework

We implemented our system on top of Storm. As shown in Figure 5, the nimbus in the Storm master node is responsible for assigning tasks to the Storm worker nodes and monitoring the failures. On receiving a job, the nimbus will transfer the query topology to the PPA plan manager, which will generate a PPA recovery plan under the constraint of resource usage of active replication. The PPA recovery plan consists of two parts: a completely passive standby plan and a partially active replication plan. Based on the PPA recovery plan, the replication manager in the worker nodes will create checkpoints to passively replicate the whole query topology. Checkpoints will be stored onto a set of standby nodes. The replication manager will create active replicas for the tasks that are included in the partially active replication plan. The active replicas can support fast failure recovery and will also be deployed onto the standby nodes.

Once a failure is detected by the nimbus, The recovery manager in the Storm master node will decide how to recover the failed tasks based on the PPA replication plan. For the tasks that are actively replicated, the recovery manager will notify the nimbus to recover them using their active replicas such that the tentative results could be produced as soon as possible. The failed tasks that are passively replicated will be recovered with their latest checkpoints.

6.2 PPA Fault Tolerance

Passive Replication. In PPA, checkpoints of the processing tasks will be periodically created and stored at the standby nodes. We adopted the batch processing approach [25] to guarantee the processing ordering of inputs during recovery is identical to that before the failure. With this approach, input tuples are divided into a consecutive set of batches. A task will start processing a batch after it receives all its input tuples belonging the current batch. This is ensured by waiting a batch-over punctuation from each of its upstream neighboring tasks. Tuples within a batch will be processed in a predefined round-robin order. The effect of batch size on the system performance has been researched in previous work [8].

A single point failure can be recovered by restarting the failed task, loading its latest checkpoint and replaying its upstream tasks' buffered data. The downstream tasks will skip the duplicated output from the recovering task until the end of the recovery phase. While recovering a correlated failure, if a task and its upstream neighboring task are failed simultaneously and its checkpoint is made later than its upstream peers', the recovery of the downstream task can only be started after its upstream peer has caught up with the processing progress. In other words, synchronizations have to be carried out among the neighboring tasks.

Active Replication. If task t has an active replica t', the output buffer of t' will store the output tuples produced by processing the same input in the same sequence as t does. The downstream tasks of t will subscribe the outputs from both t and t'. By default, the output of t' is turned off. To reduce the buffer size on t', its primary, t, will periodically notify t' about the latest output progress and the latter can then trim its output buffer. If t is failed, t' will start sending data to the downstream tasks of t. The downstream tasks will eliminate the duplicated tuples from t' by recognizing their sequence numbers. The batch processing strategy can guarantee an identical processing order between the primary and active replica of a task.

Tentative Outputs. As checkpoint-based recovery requires replaying the buffered data and synchronizations among the connected tasks and hence incurs significant recovery latency, PPA has the option to continue producing tentative results once the actively replicated tasks are recovered. Recall that during normal processing, a task will only start processing a batch after receiving the batch-over punctuations from all of its upstream neighboring tasks. If any of its upstream neighboring tasks fails, the recovery manager in the Storm master node will generate the necessary batch-over punctuations for those failed tasks, such that a batch could be processed without the inputs from the failed tasks and tentative outputs will be generated with an incomplete batch. After the failed tasks are recovered, the recovery manager will stop sending the batch-over messages for them such that the downstream tasks will wait for the batch contents from the recovered tasks before processing a batch. After all the failed tasks are recovered, the topology will start generating accurate outputs.

In this paper, we assume the adoption of similar techniques proposed in [4] to reconcile the computation state and correct the tentative outputs and leave the implementation of these techniques as our future work.

7. EVALUATION

The experiments are run over the Amazon EC2 platform. We build a cluster consisting of 36 instances, of which 35 m1.medium instances are used as the processing nodes and one c1.xlarge instance is set as the Storm master node. Heartbeats are used to detect node failures in a 5-second interval. The recovery latency is calculated as the time interval between the moment that the failure is detected and the instant when the failed task is recovered to its processing progress before failure. The processing progress of a task is defined as a vector. Each field of the progress vector contains the sequence number of the latest processed tuple from a specific input stream of the task. A failed task is marked as recovered if the values of all the fields in its current progress vector are larger than or equal to the values of the corresponding fields of the progress vector before failure. Additional information of the experiment configuration will be presented in the following sections.

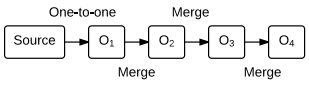


Figure 6: Topology used in the experiments of recovery efficiency in the scale of operator.

7.1 Recovery Efficiency

In the first set of experiments, we study the recovery efficiencies of different fault-tolerance techniques, including active replication, checkpoint and the default fault-tolerance technique in Storm [2]. In Storm, if failure happens, the source data will be reprocessed from scratch through the whole topology to rebuild the states of the tasks.

We implement a topology that consists of 1 source operator and 4 synthetic operators. The structure of this topology is depicted in Figure 6. The source operator consists of totally 16 tasks, which are averagely deployed on 4 nodes. All of the source tasks produce input tuples for their downstream neighboring tasks in a specified rate (1000 tuples/s or 2000 tuples/s). The degree of parallelization of operators O_1, O_2, O_3 and O_4 are set as 8, 4, 2 and 1 respectively. Each task in O_1 receives inputs from two source tasks and each task in O_2, O_3 and O_4 receives inputs from two upstream neighboring tasks. The primary replicas of the 15 synthetic tasks are evenly distributed among the 15 nodes. In addition, there are another 15 nodes used as the backup nodes to store the checkpoints and to run the active replicas.

Each of the four synthetic operators maintains a sliding window whose sliding step is set as 1 second and window interval varies from 10 seconds to 30 seconds. The state of each task of a synthetic operator is composed by the input data within the current window interval. The largest state size of a task is equal to the result of the input rate multiplies the window interval. The selectivity of the synthetic operator is set as 0.5. Therefore a task produces one tuple for every two input tuples that it has processed.

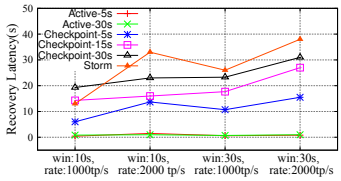


Figure 7: Recovery latency of single node failure.

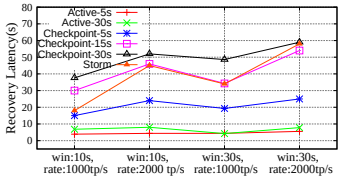


Figure 8: Recovery latency of correlated failure.

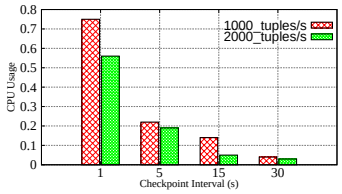


Figure 9: Resource usage of maintaining checkpoints, window length: 30 seconds.

Single Node Failure. Figure 7 presents the recovery latency of single node failures with various input rates and window intervals using different fault-tolerance techniques. For active replication, we vary the intervals of trimming the output buffer of a task replica, which is equivalent to the frequency of synchronizing the replica with its primary task. One can see that the active approach has much lower recovery latency than the passive approaches and the changes of window intervals and input rates have little influence. On the other hand, the recovery latency with both Checkpoint and Storm increase proportionally with the input rate. This is because a higher input rate results in more tuples to be replayed during recovery for both approaches. Furthermore, one can see that the recovery latency with checkpoints increases with the checkpoint interval. This is because the number of tuples that need be reprocessed to recover the task state will increase with the checkpoint interval.

As Storm will have to replay more source data with longer window intervals, one can see that the recovery latencies of Storm with 30-second windows are higher than those with 10-second windows. Another factor that influences the recovery latency of Storm is the location of the failed task in

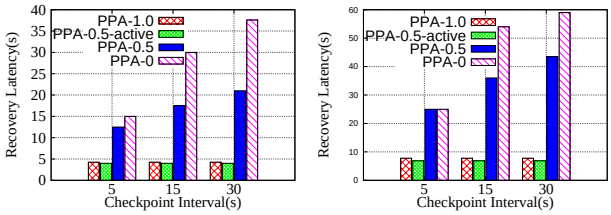
the topology, because the replayed tuples will be processed by all the tasks located between the tasks of the source operator and the failed tasks. Thus the recovery latencies of Storm are higher than that of the checkpoint approach in most of the cases in this experiment. Here, we record the recovery latencies of tasks in different locations within the topology in Storm and report their average values.

Correlated Failure. We inject a correlated failure by killing all the nodes on which the primary replicas of the tasks are deployed. We present the recovery latencies of correlated failures in Figure 8. One can see that active replication has much lower recovery latencies than Checkpoint and Storm. Compared to active replication with a 30-second synchronization period, setting the synchronization period as 5 seconds leads to faster failure recoveries. This is because, with a longer synchronization period, an active replica will send more buffered tuples to its downstream tasks if its primary fails. The difference is limited though, because the downstream tasks can eliminate the duplicated inputs in a relatively efficient way by recognizing the input tuples' sequence numbers. On the other hand, one can see that the recovery latencies of Checkpoint increase rapidly with the increase of input rates and checkpoint intervals. The recovery latencies of Storm are lower than that of Checkpoint with a 30-second checkpoint interval. This is because the window intervals in this set of experiments are relatively short. In Storm, to build the window states, all the sources tuples belonging to the unfinished window instances in the failed tasks will be replayed, whose number increases linearly with the window length. While for Checkpoint, the number of input tuples that should be reprocessed to recover a failed task is at most equal to the value of the input rate multiplies the checkpoint interval.

By comparing the recovery latencies presented in Figure 7 and Figure 8, it can be seen that the recovery latencies with active replication are lower than the passive approaches and are relatively stable under the scenarios of various input rates and window intervals. Moreover, the benefits of using active replication are larger in the case of correlated failure than in the case of single node failure. This is because checkpoints for different nodes are often made asynchronously and some synchronization operations will be performed during the recovery of correlated failures.

The latency of failure recovery with checkpoint can be reduced by setting a short checkpoint interval. But the resource usage of maintaining checkpoints varies with different checkpoint intervals. Figure 9 presents the ratio of the CPU usage of maintaining checkpoint to that of normal computation within a task. We can see that the CPU usage of maintaining checkpoints increases quickly with shorter checkpoint intervals and making checkpoint with very short intervals such as one second is prohibitively expensive. Although active replication consumes more recourses than the passive approach, the low-latency recovery of active replication makes it meaningful in the context of MPSPEs.

Recovery with PPA. We conducted experiments to study the performance of PPA with three active replication plans denoted as PPA-1.0, PPA-0.5 and PPA-0 respectively. These PPA plans consume various amount of resources for active replication. In PPA-1.0, all the tasks in the topology will be actively replicated. PPA-0.5 is a hybrid replication plan where only half of the tasks have active replica. The third plan, PPA-0, is a purely passive replication plan where all



(a) rate:1000 tuples / sec (b) rate:2000 tuples / sec

Figure 10: Recovery latency of a correlated failure with PPA, window length : 30 seconds. PPA-0.5-active indicates the recovery latency of actively replicated tasks in plan PPA-0.5.

the tasks are only replicated with checkpoint. The results are presented in Figure 10. As the failed tasks with active replicas will be recovered faster than those using checkpoints, the overall recovery latencies of PPA-0.5 are higher than that of PPA-1.0 but lower than that of PPA-0. Note that with PPA-0.5, the recovery latencies of tasks with active replicas (denoted as PPA-0.5-active in Figure 10) are much lower than that of recovering all the failed tasks (denoted as PPA-0.5 in Figure 10). One can also see that the recoveries of PPA-0.5-active consume slightly less time than PPA-1.0, this is because the number of actively replicated tasks recovered in PPA-0.5-active is only the half of that in PPA-1.0. This set of experiments illustrate that the purely active replication plan outperforms the hybrid and purely passive plan regarding the recovery latency. With a hybrid plan, as the recoveries of actively replicated tasks will be finished earlier than that of the passively replicated ones, PPA can generate tentative outputs without waiting for the slow recoveries of passively replicated tasks.

7.2 Tentative Output Quality

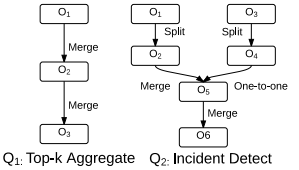


Figure 11: *Top-k* aggregate query(Q_1) and incident detect query(Q_2) in the scale of operator.

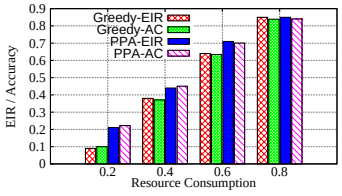


Figure 12: Comparing the values of EIR and the query accuracy. Query: top-k aggregate.

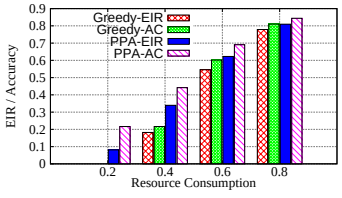
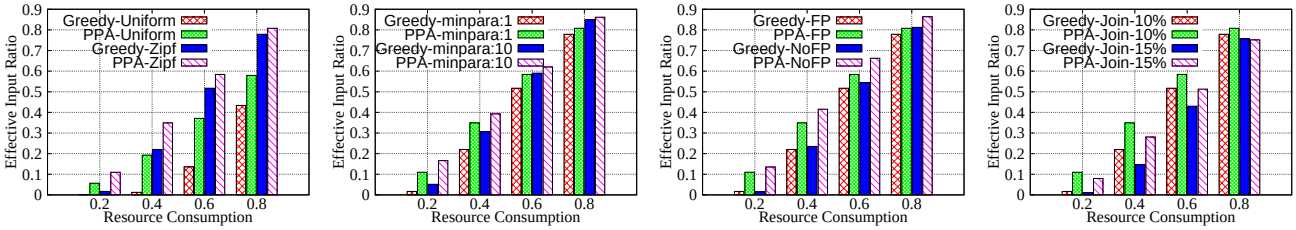


Figure 13: Comparing the values of EIR and the query accuracy. Query: incident detect.

Validation of the EIR metric. In this set of experiments, we will examine whether the EIR metric can some-



(a) Workload skewness (b) Degree of parallelization (c) Full partitioning (d) Fraction of join operators

Figure 14: Comparing EIR of PPA and greedy algorithm with random topologies of various specifications, number of operators is set as a random integer between 5 and 10. (a): The workloads of tasks within an operator are distributed in uniform or Zipfian distribution (with parameter $s = 0.1$). (b): The minimal degree of operator parallelization is set as 1 or 10, the maximal value is 20. (c): Topologies with or without full partitioning method(d): The fraction of join operators in the topologies is set as 10% or 15%.

how predict the actual quality of the tentative output. We implement two sliding window queries whose inputs are, respectively, from real and synthetic datasets. For each query, we define a query accuracy function based on the query semantics. We process the queries with multiple replication plans of various active replication ratios and compare the actual output accuracy with the EIR values after correlated failures. Due to the prohibitive complexity of the dynamic algorithm, we cannot complete it for our experiments within a reasonable time so we do not include it here.

The first query is a sliding-window query that calculates the *top-50* hottest entries of the official website of World Cup 1998. The window length is set as 5 minutes and the sliding step is set as 10 seconds. The input dataset of this query is the server access log of the official website for the World Cup 1998 made during the entire day of June 29, 1998 [1]. There are in total $62,228,636$ access records in this log and each record has eight fields including the access time stamp, client id, requested entry id and the server id. In the experiments, we replay the raw input stream in a rate which is 48 times faster than the original data rate. We implement this query as a topology that conducts the computation of hierarchical aggregate, which is a common computation in data stream applications. The topology consists of 3 operators: O_1, O_2 and O_3, whose degree of parallelization are set as 30, 4 and 1 respectively. All of these operators use the Merge partitioning method. The structure of this topology is depicted as Q_1 in Figure 11. Input tuples are partitioned to the tasks in O_1 by their server ids. Tasks in O_1 split the input stream into a set of consecutive 1-second window instances and calculate their aggregate results. Tasks in O_2 merge the aggregate results of the 1-second window instances from O_1 and send the merge results to the single task in O_3, which generates the final query outputs.

By denoting the tentative outputs as S_T and the accurate outputs of Q_1 as S_A, we define the query accuracy of Q_1 as: $\frac{|S_T \bigcap S_A|}{|S_A|}$. One can see that, in Figure 12, by increasing the ratio of actively replicated tasks in the topology, the EIR and accuracy of the tentative outputs increase at a similar pace. This shows that the typical top-k query conforms to our assumption that the accuracy of the query output increases with more complete input. Furthermore, as the active replicas selected by the greedy algorithm may not be able to form complete MC-trees within the topology, PPA consistently outperforms the Greedy algorithm especially when the active replication ratio is low.

The second query is a sliding-window query that detects the traffic incidents resulting in traffic jams. The window interval is 5 minutes and the sliding step is 10 seconds. As relevant datasets for this query are usually not publicly available due to privacy considerations, we generate a synthetic dataset in a community-based navigation application. There are two streams in this dataset: the user-location stream and the incident stream. The rate of the user-location stream is set as 20,000 tuples per second, which consists of location events containing the time, speed and location of the users. The incident stream is composed of user-reported incident events that contains the time stamp of incident, location and incident type. The time interval between two consecutive incidents is set as 2 seconds. We distribute 100,000 users among 1000 virtual road segments following the Zipfian distribution (with parameter $s = 0.5$). The incident probability of a segment is set to be proportional to the number of users located on it. If an incident occurs on a segment, all the users on this segment will report an incident event.

The topology of this query is presented as Q_2 in Figure 11. Q_2 consists of 6 operators: O_1, O_2, O_3 O_4, O_5 and O_6, whose degree of parallelization are set as 4, 20, 1, 5, 5 and 1 respectively. Tasks in O_1 and O_3 generate the user-location and incident streams by loading the pre-generated dataset. Tasks in O_3 calculate the average speed of each segment every 1 second. Tasks in O_4 combine the user-reported incident events into distinct incident events. O_5 is a join operator, which joins the segment-speed stream from O_2 and the distinct-incident stream from O_4. The outputs of tasks in O_5 are the incidents that incur traffic jams. There is only one task in O_6, which computes the aggregate of the outputs from tasks in O_5.

The accuracy function of this query is defined as $\frac{|I_T \bigcap I_A|}{|I_A|}$, where I_T is the set of tentative incidents generated with correlated failure and I_A is the set of accurate incidents generated without failure. Results of this set of experiments are presented in Figure 13. One can see that, although the accuracy values turn out to be larger than the values of EIR, they both increase in a similar pace with the increase of actively replication ratio. With a smaller active replication ratio, the advantages of PPA in terms of both EIR values and actual accuracies in comparing to the greedy algorithm get larger. While the active replication ratio is set as 0.2, as the amount of resources is not enough to actively replicate any MC-tree completely in this case, the replication plan generated by the greedy algorithm can not make any tentative outputs.

Random synthetic topology. To conduct a comprehensive performance study of PPA with various types of topologies, we implement a random topology generator which can generate topologies with different specifications including

the number of operators, the degree of operator parallelization, the distribution of operator partitioning methods, the fraction of join operators in the topology and the workload distribution of tasks within an operator. In the experiments, for each set of topology specifications, we generate 100 synthetic topologies and use them as the inputs of PPA and the greedy algorithm to compare their performances in terms of EIR. As we cannot derive the actual output accuracies for these randomized topologies, we do not use them here.

In Figure 14, one can see that, PPA outperforms the greedy algorithm in all the combinations of topology specifications and active replication ratios. While the replication ratios are small, the differences of EIR between PPA and the greedy algorithm are larger than those with large active replication ratios as the greedy algorithm is agnostic to the structure of the query topologies.

Figure 14(a) depicts the effects of workload skewness of tasks within the operators. We can see that PPA has better performance for topologies that have higher skewness of task workloads. This is because, as the skewness of tasks workloads increases, the skewness of MC-trees' contributions to the value of EIR increases and PPA prioritizes replicating tasks that are in the MC-trees leading to higher EIR. In Figure 14(b), we report the results with the minimum parallelization degree of an operator as 1 and 10. One can see that increasing the degree of operator parallelization will also increase the value of EIR, as increasing the degree of operator parallelization slightly increases the skewness of the workload of tasks within operators. In Figure 14(c), the EIR of topologies without using Full partitioning is higher than those that have operators with Full partitioning. This is because within an operator using Full partitioning, the failure of any task will reduce the input of all the downstream tasks. Figure 14(d) presents the results with various fractions of operators being join operators. EIR decreases with more operators in the topologies are join operators. This is because with more join operators, MC-trees will contain tasks from more source operators due to the correlation brought by the join operators and therefore, the failure of a task incurs input loss of more source operators.

8. CONCLUSION

In this paper we present a passive and partially active (PPA) fault-tolerance scheme for MPSPEs. In PPA, passive checkpoints are used to provide fault-tolerance for all the tasks, while active replications are only applied to selective ones according to the availability of resources. A partial active replication plan is optimized to maximize the accuracy of tentative outputs during failure recovery. The experimental results indicate that upon a correlated failure, PPA can start producing tentative outputs up to 10 times faster than the completion of recovering all the failed tasks. Hence PPA is suitable for applications that prefer tentative outputs with minimum delay. The experiments also show that our structure-aware algorithms can achieve up to one order of magnitude improvements on the qualities of tentative outputs in comparing the greedy algorithm that is agnostic to query topology structures, especially when there is limited resource available for active replications. Therefore, to optimize PPA, it is critical to take advantage of the knowledge of the query topology's structure.

9. REFERENCES

[1] `http://ita.ee.lbl.gov`.

[2] `http://www.storm-project.net`.

[3] B. Babcock, M. Datar, et al. Load shedding for aggregation queries over data streams. ICDE'04.

[4] M. Balazinska, H. Balakrishnan, et al. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst*, 2008.

[5] P. Bellavista, A. Corradi, et al. Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. In *EDBT'14*.

[6] A. Biem, E. Bouillet, et al. Ibm infosphere streams for scalable, real-time, intelligent transportation services. SIGMOD'10.

[7] F. Castro, M. Raul, et al. Integrating scale out and fault tolerance in stream processing using operator state management. SIGMOD'13.

[8] T. Das, Y. Zhong, et al. Adaptive stream processing using dynamic batch sizing. *Master Thesis'2014*.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] Y. Gu, Z. Zhang, et al. An empirical study of high availability in stream processing systems. Middleware'09.

[11] T. Heath, R. P. Martin, et al. Improving cluster availability using workstation validation. SIGMETRICS'02.

[12] J.-H. Hwang, U. Cetintemel, and others. Fast and highly-available stream processing over wide area networks.

[13] J.-H. Hwang et al. High-availability algorithms for distributed stream processing. ICDE'05.

[14] J.-H. Hwang, Y. Xing, et al. A cooperative, self-configuring high-availability solution for stream processing. ICDE'07.

[15] M. Isard, M. Budiu, et al. Dryad: Distributed data-parallel programs from sequential building blocks. EuroSys '07.

[16] G. Jacques-Silva, B. Gedik, et al. Fault injection-based assessment of partial fault tolerance in stream processing applications. DEBS'11.

[17] J. Kang, J. F. Naughton, et al. Evaluating window joins over unbounded streams. ICDE'03.

[18] Y. Kwon, M. Balazinska, et al. Fault-tolerant stream processing using a distributed, replicated file system.

[19] A. Martin, C. Fetzer, and A. Brito. Active replication at (almost) no cost. SRDS'11.

[20] S. Nath, H. Yu, Gibbons, et al. Subtleties in tolerating correlated failures in wide-area storage systems. NSDI'06.

[21] B. Nikhil, B. Ranjita, et al. Towards optimal resource allocation in partial-fault tolerant applications. In *INFOCOM'08*.

[22] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. DSN'06.

[23] L. Su and Y. Zhou. Unplublished manuscript. `http://imada.sdu.dk/~lsu/NP_proof.pdf`.

[24] P. Upadhyaya, Y. Kwon, et al. A latency and fault-tolerance optimizer for online parallel query plans. SIGMOD'11.

[25] M. Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. SOSP '13.

[26] Z. Zhang, Y. Gu, et al. A hybrid approach to high availability in stream processing systems. ICDCS '10.

[27] J. Zhou et al. Advanced partitioning techniques for massively distributed computation. SIGMOD'12.